Sparse supernodal solver with low-rank compression for solving the frequency-domain Maxwell equations discretized by a high order HDG method

Grégoire Pichon, Eric Darve, Mathieu Faverge, Stéphane Lanteri, Pierre Ramet, Jean Roman

Sparse supernodal solver with low-rank compression for solving the frequency-domain Maxwell equations discretized by a high order HDG method

November 20th, 2017 - JJC

G. Pichon [a], E. Darve [b], M. Faverge [a], S. Lanteri [c], P. Ramet [a], J. Roman [a]

[a]Inria, Bordeaux INP, CNRS, University of Bordeaux, [b]Stanford University,
[c]Inria, Université Nice - Sophia Antipolis, CNRS

Introduction

PaStiX: Sparse Direct Solver

- Originally designed for distributed heterogeneous architectures
- Recent works introduced low-rank compression to reduce the burden on time and memory complexities

HORSE: HDG solver

- Solve a reduced system on faces instead of elements
- Lead to a sparse system smaller than the one using CG or DG
- Domain-Decomposition is then used
- Use a sparse direct solver on each subdomain

Objective for this talk: evaluate the impact of using low-rank compression to enhance subdomains' factorization in HORSE.

Outline

1. Background on Sparse Direct Solvers
2. Block Low-Rank Operations
3. Numerical Experiments
4. Case study: PaStiX inside a HDG method

1

Background on Sparse Direct Solvers

PaStiX: a sparse direct solver

Current sparse direct solvers for 3D problems

- $\Theta(n^2)$ time complexity
- $\Theta(n^{\frac{4}{3}})$ memory complexity
- BLAS 3 operations: efficient operations

Block Low-Rank solver

- In many applications, the machine precision is not required
- **Minimal Memory** strategy allows to save memory
- **Just-In-Time** strategy allows to reduce time-to-solution

Objective: build an algebraic low-rank solver following the supernodal approach of PaStiX, with block-data structures

Sparse Direct Solvers approach

Problem: solve $Ax = b$

- Cholesky: factorize $A = LL^t$ (symmetric pattern $(A + A^t)$ for LU)
- Solve $Ly = b$
- Solve $L^t x = y$

PaStiX steps

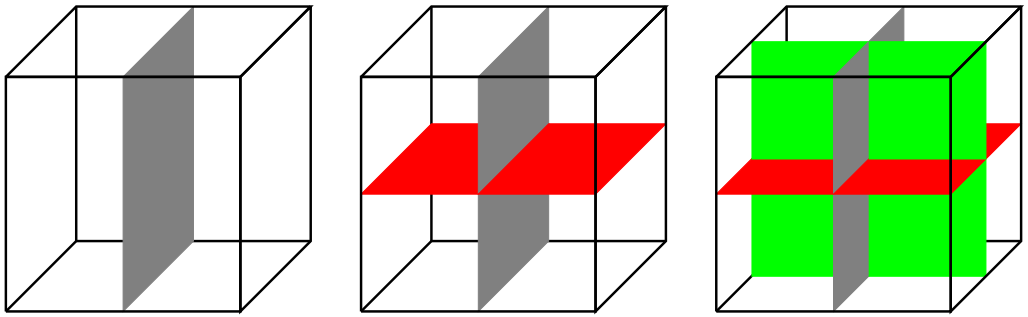
1. Order unknowns to minimize the fill-in
2. Compute a symbolic factorization to build L structure
3. Factorize the matrix in place on L structure
4. Solve the system with forward and backward triangular solves

Ordering with Nested Dissection

Entry graph $G = (V, E, \sigma_p)$
V: vertices, E: edges, σ_p: unknowns permutation

Algorithm to compute σ_p

1. Partition $V = A \cup B \cup C$
2. Order C with larger numbers: $V_A < V_C$ and $V_B < V_C$
3. Apply the process recursively on A and B



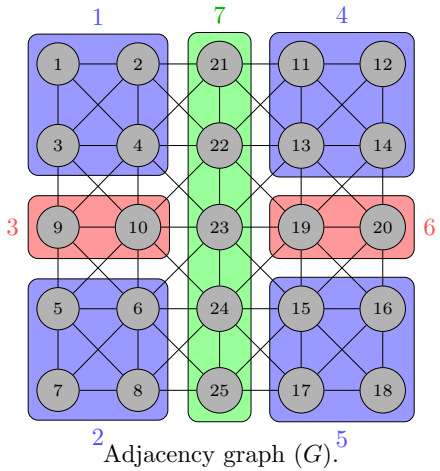
Three-levels of nested dissection on a regular cube.

Metis and Scotch are mostly used to perform this partitioning.

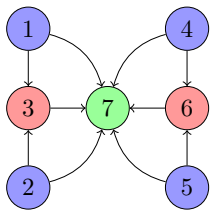
Symbolic Factorization

General approach

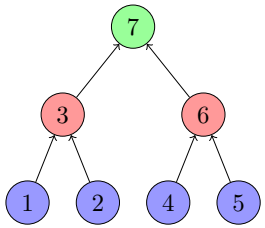
1. Build a partition with the nested dissection process
2. Compress information on data blocks
3. Compute the block elimination tree using the block quotient graph



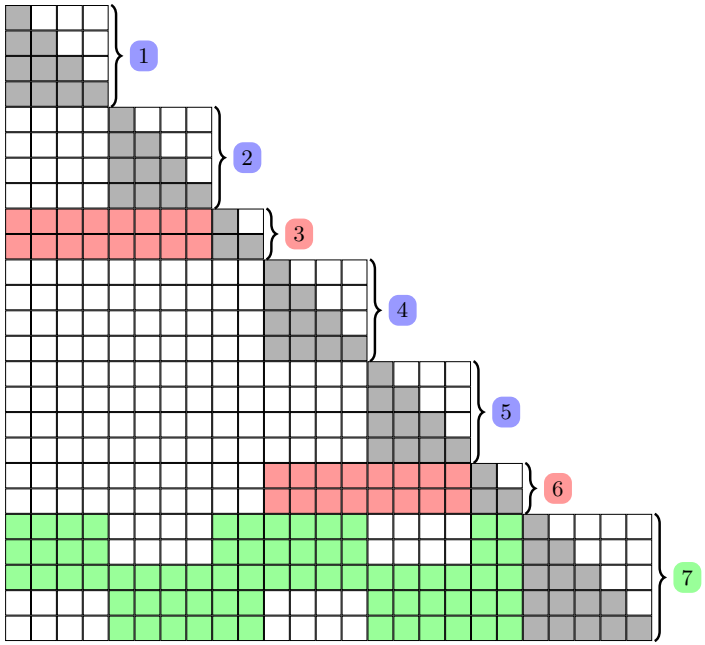
Adjacency graph (G).



Quotient graph (G^*/P) $= (G/P)^*$



Elimination tree (T).

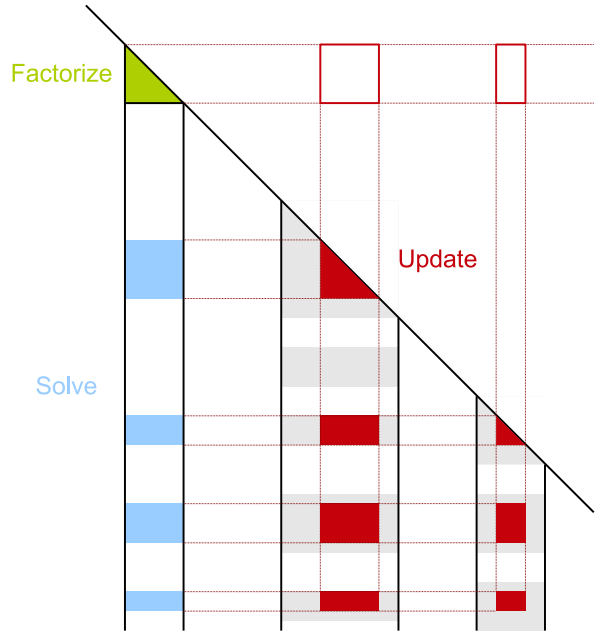


Factorized matrix (L).

Numerical Factorization

Algorithm to eliminate the column block k

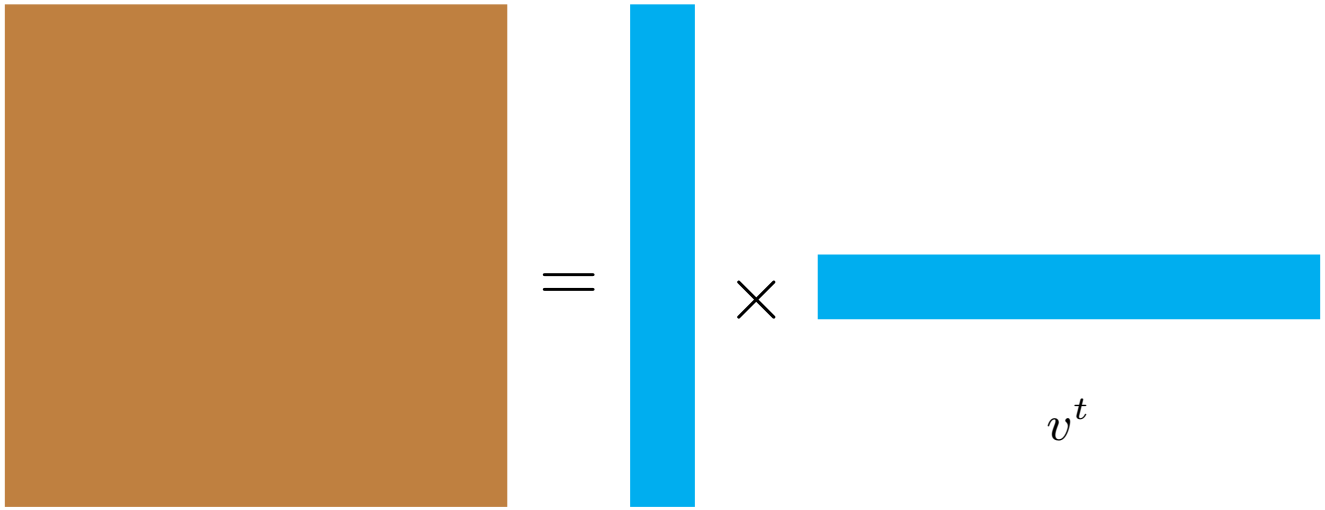
1. Factorize the diagonal block (POTRF/GETRF)
2. Solve off-diagonal blocks in the current column (TRSM)
3. Update the trailing matrix with the column's contribution (GEMM)



2

Block Low-Rank Operations

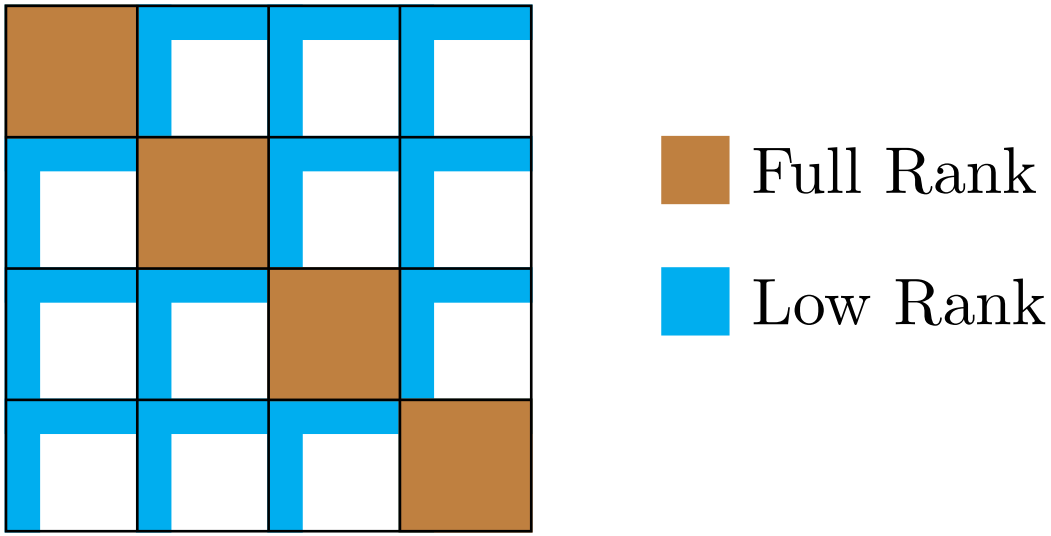
Low-Rank Compression



$$M \in \mathbb{R}^{n \times n}; u, v \in \mathbb{R}^{n \times r}$$

Memory Consumption reduced: $2nr$ instead of n^2
Several methods can be used: SVD, RRQR, BDLR, ACA...
The objective is to compute $\hat{A}$ such that $||A - \hat{A}||_2 < \tau$

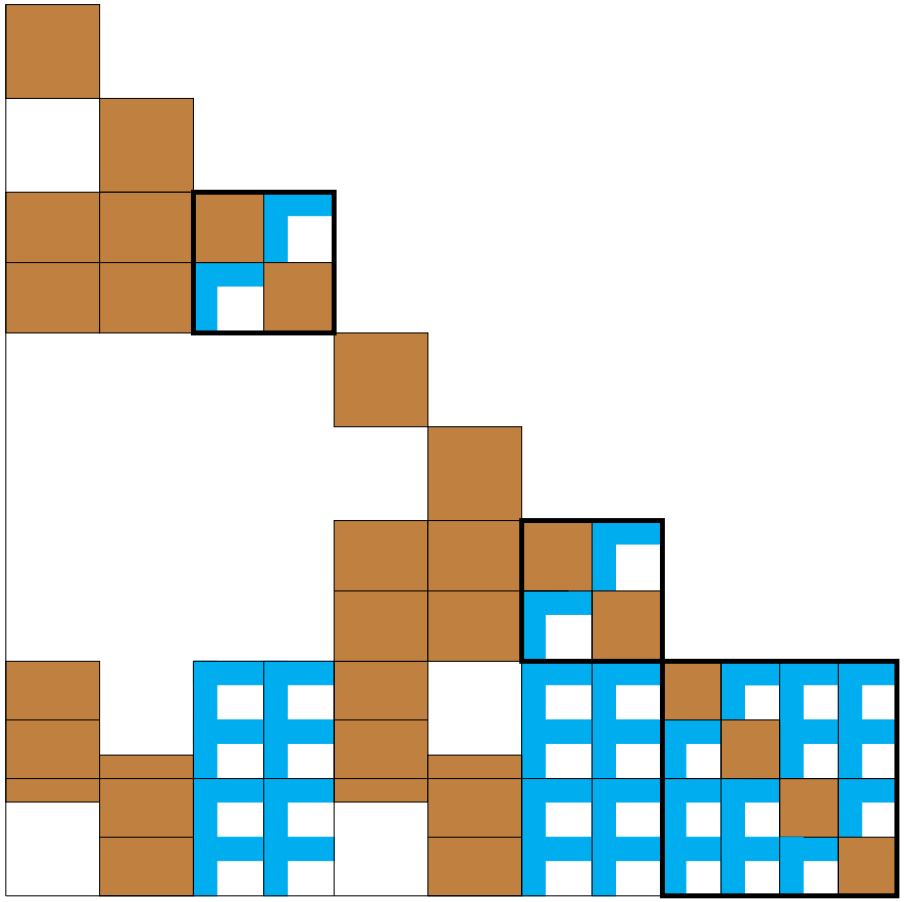
Block Low-Rank Compression for a dense matrix



Some related works for dense matrices

- LSTC
- Airbus (hmat-oss)
- CEA

Block Low-Rank Compression – Symbolic Factorization



Large off-diagonal are low-rank, in the form uv^t

Some Related Works

Sparse Block Low-Rank solver: BLR-MUMPS

- They introduced an approach to reduce the time-to-solution
- The main difference is the difference between multifrontal and supernodal

Some other approaches for sparse

- Strumpack HSS by Ghysels et al. uses randomized sampling to perform an efficient extend-add
- HODLR by Darve et al. uses pre-selection of rows and columns (BDLR)
- $\mathcal{H}$-LU by Hackbusch et al. does not fully exploit the symbolic factorization
- HODLR/HSS like method by Chadwick et al. uses estimation of ranks and the geometry of the problem

Block Low-Rank Algorithm

Approach

- Large supernodes are partitioned into a set of smaller supernodes
- Large off-diagonal blocks are represented as low-rank blocks

Operations

- Diagonal blocks are dense
- TRSM are performed on low-rank off-diagonal blocks
- GEMM are performed between low-rank off-diagonal blocks. It creates contributions to dense or low-rank blocks: this is the extend-add problem

Compression techniques

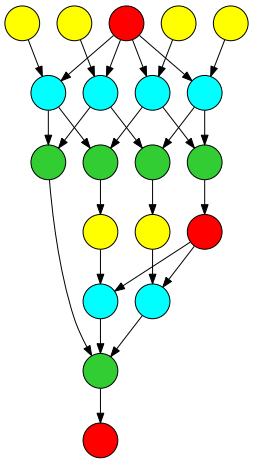
- SVD, RRQR for now
- Possible extension to any algebraic method: ACA, randomized techniques...

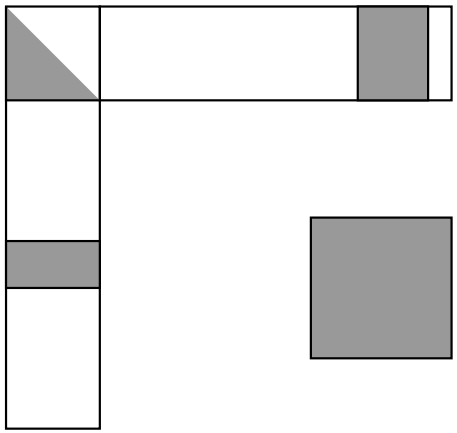
Strategy Just-In-Time

Compress L

1. Eliminate each column block
   - **1.1** Factorize the dense diagonal block
     Compress off-diagonal blocks belonging to the supernode
   - **1.2** Apply a TRSM on LR blocks (cheaper)
   - **1.3** LR update on dense matrices (**LR2GE** extend-add)
2. Solve triangular systems with low-rank blocks

Compression
GETRF (Facto)
TRSM (Solve)
LR2LR (Update)
LR2GE (Update)



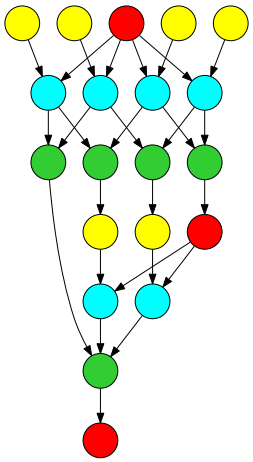


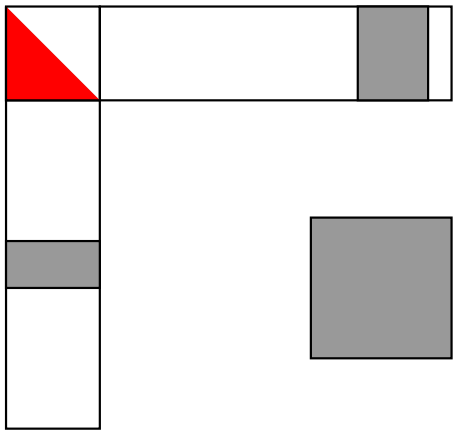
Strategy Just-In-Time

Compress L

1. Eliminate each column block
   - **1.1** Factorize the dense diagonal block
     Compress off-diagonal blocks belonging to the supernode
   - **1.2** Apply a TRSM on LR blocks (cheaper)
   - **1.3** LR update on dense matrices (**LR2GE** extend-add)
2. Solve triangular systems with low-rank blocks

Compression
GETRF (Facto)
TRSM (Solve)
LR2LR (Update)
LR2GE (Update)



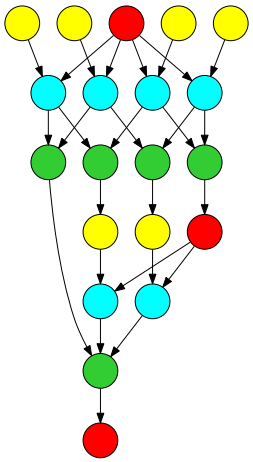


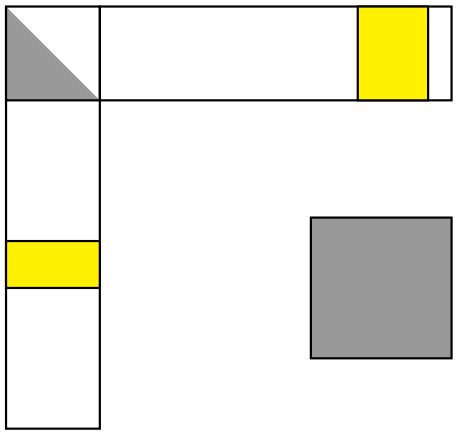
Strategy Just-In-Time

Compress L

1. Eliminate each column block
   - **1.1** Factorize the dense diagonal block
     Compress off-diagonal blocks belonging to the supernode
   - **1.2** Apply a TRSM on LR blocks (cheaper)
   - **1.3** LR update on dense matrices (**LR2GE** extend-add)
2. Solve triangular systems with low-rank blocks

Compression
GETRF (Facto)
TRSM (Solve)
LR2LR (Update)
LR2GE (Update)



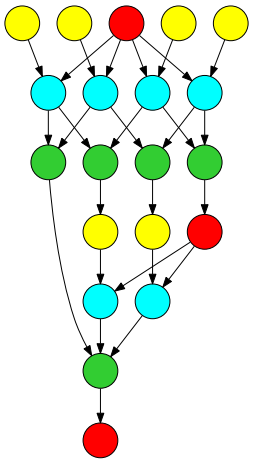


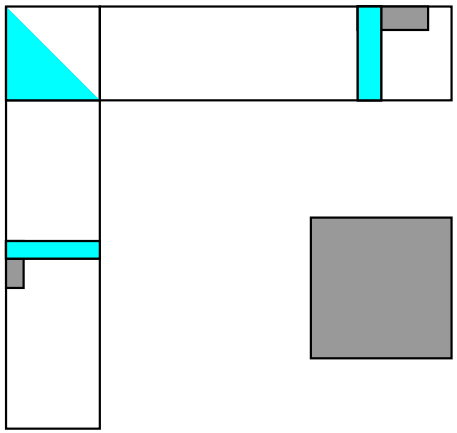
Strategy Just-In-Time

Compress L

1. Eliminate each column block
   - **1.1** Factorize the dense diagonal block
     Compress off-diagonal blocks belonging to the supernode
   - **1.2** Apply a TRSM on LR blocks (cheaper)
   - **1.3** LR update on dense matrices (**LR2GE** extend-add)
2. Solve triangular systems with low-rank blocks

Compression
GETRF (Facto)
TRSM (Solve)
LR2LR (Update)
LR2GE (Update)



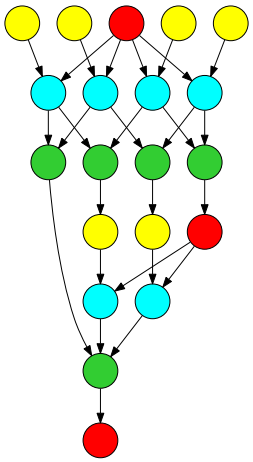


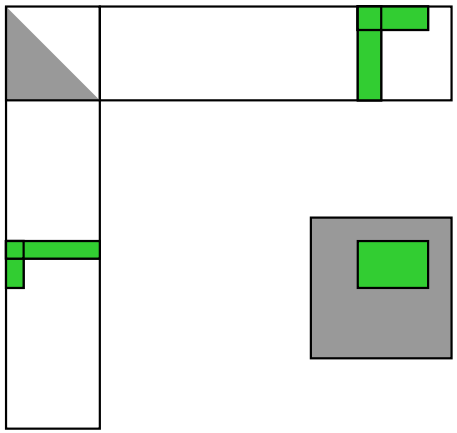
Strategy Just-In-Time

Compress L

1. Eliminate each column block
   - **1.1** Factorize the dense diagonal block
     Compress off-diagonal blocks belonging to the supernode
   - **1.2** Apply a TRSM on LR blocks (cheaper)
   - **1.3** LR update on dense matrices (**LR2GE** extend-add)
2. Solve triangular systems with low-rank blocks

Compression
GETRF (Facto)
TRSM (Solve)
LR2LR (Update)
LR2GE (Update)



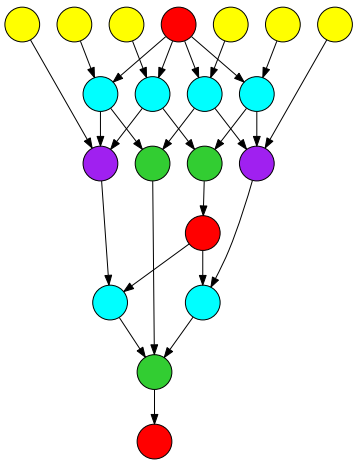


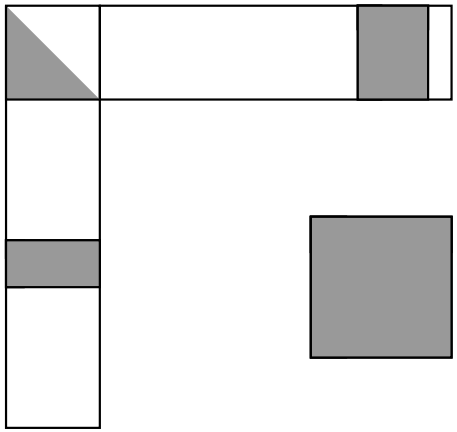
Strategy Minimal Memory

Compress A

1. Compress large off-diagonal blocks in A (exploiting sparsity)
2. Eliminate each column block
   - **2.1** Factorize the dense diagonal block
   - **2.2** Apply a TRSM on LR blocks (cheaper)
   - **2.3** LR update on LR matrices (**LR2LR** extend-add)
3. Solve triangular systems with LR blocks

	Compression
	GETRF (Facto)
	TRSM (Solve)
	LR2LR (Update)
	LR2GE (Update)



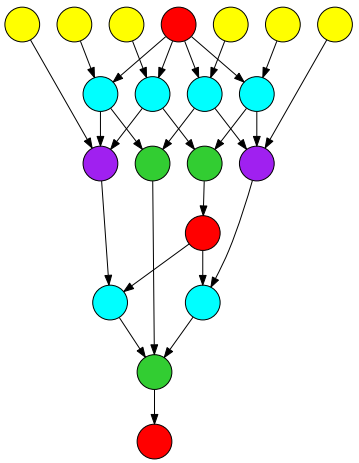


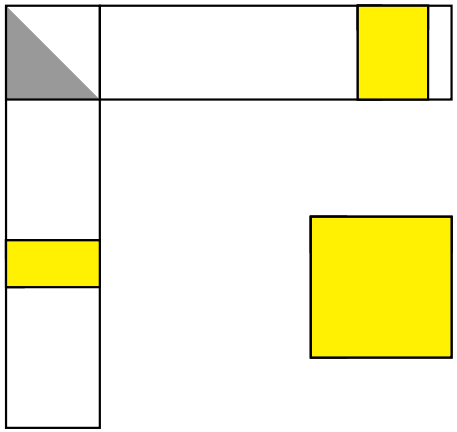
Strategy Minimal Memory

Compress A

1. Compress large off-diagonal blocks in A (exploiting sparsity)
2. Eliminate each column block
   - **2.1** Factorize the dense diagonal block
   - **2.2** Apply a TRSM on LR blocks (cheaper)
   - **2.3** LR update on LR matrices (**LR2LR** extend-add)
3. Solve triangular systems with LR blocks

	Compression
	GETRF (Facto)
	TRSM (Solve)
	LR2LR (Update)
	LR2GE (Update)



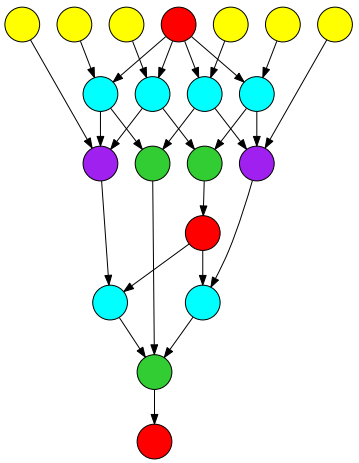


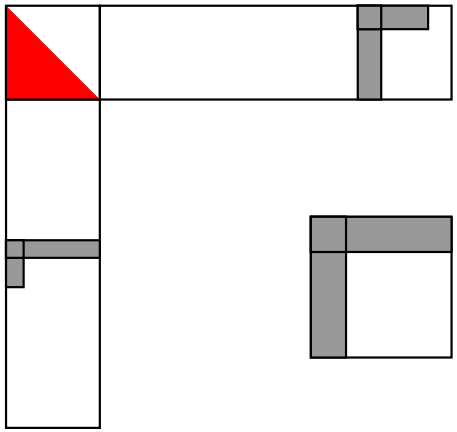
Strategy Minimal Memory

Compress A

1. Compress large off-diagonal blocks in A (exploiting sparsity)
2. Eliminate each column block
   - **2.1** Factorize the dense diagonal block
   - **2.2** Apply a TRSM on LR blocks (cheaper)
   - **2.3** LR update on LR matrices (**LR2LR** extend-add)
3. Solve triangular systems with LR blocks

	Compression
	GETRF (Facto)
	TRSM (Solve)
	LR2LR (Update)
	LR2GE (Update)



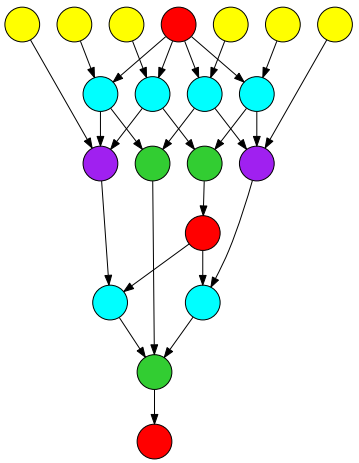


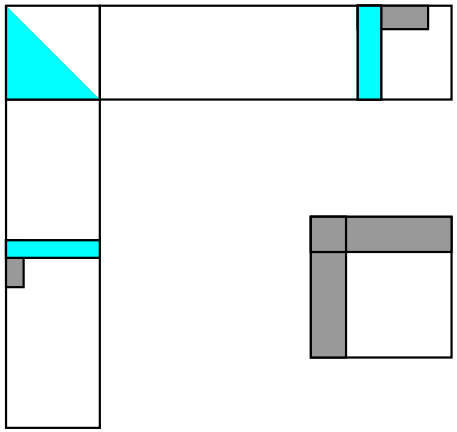
Strategy Minimal Memory

Compress A

1. Compress large off-diagonal blocks in A (exploiting sparsity)
2. Eliminate each column block
   - **2.1** Factorize the dense diagonal block
   - **2.2** Apply a TRSM on LR blocks (cheaper)
   - **2.3** LR update on LR matrices (**LR2LR** extend-add)
3. Solve triangular systems with LR blocks

	Compression
	GETRF (Facto)
	TRSM (Solve)
	LR2LR (Update)
	LR2GE (Update)



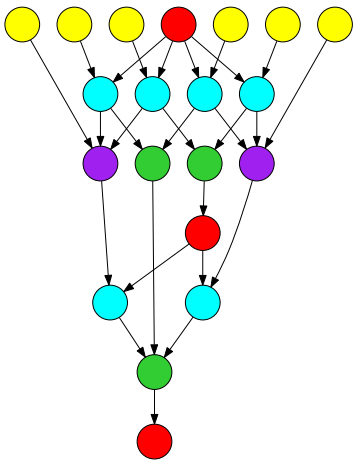


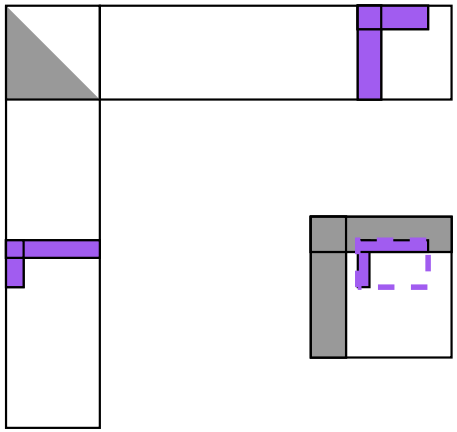
Strategy Minimal Memory

Compress A

1. Compress large off-diagonal blocks in A (exploiting sparsity)
2. Eliminate each column block
   - **2.1** Factorize the dense diagonal block
   - **2.2** Apply a TRSM on LR blocks (cheaper)
   - **2.3** LR update on LR matrices (**LR2LR** extend-add)
3. Solve triangular systems with LR blocks

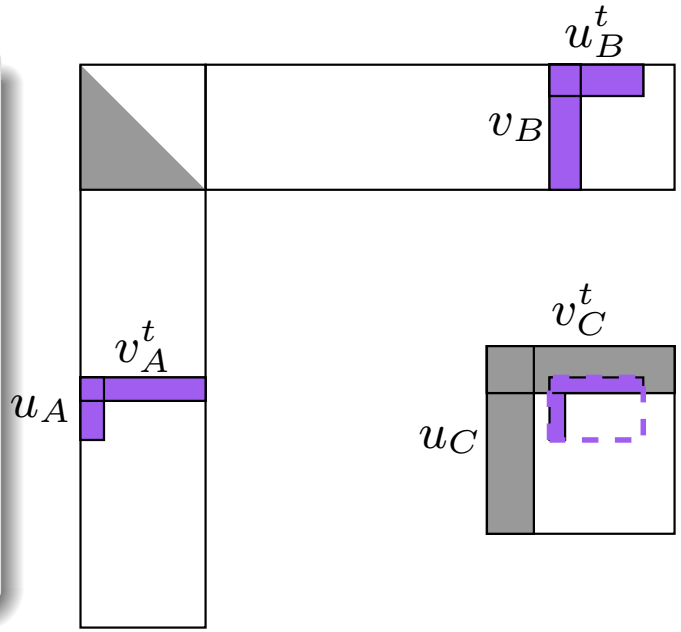
	Compression
	GETRF (Facto)
	TRSM (Solve)
	LR2LR (Update)
	LR2GE (Update)





Focus on the LR2LR kernel: build the $u_{AB}v_{AB}^t$ contribution

- Update of C with contribution from blocks A and B
- The low-rank matrix $u_{AB}v_{AB}^t$ is added to $u_Cv_C^t$
- $u_{AB}v_{AB}^t = (u_A(v_A^tv_B))u_B^t$ or $u_{AB}v_{AB}^t = u_A((v_A^tv_B)u_B^t)$
- Eventually, recompression of $(v_A^tv_B)$



Focus on the LR2LR kernel: add the contribution into $u_C v_C^t$

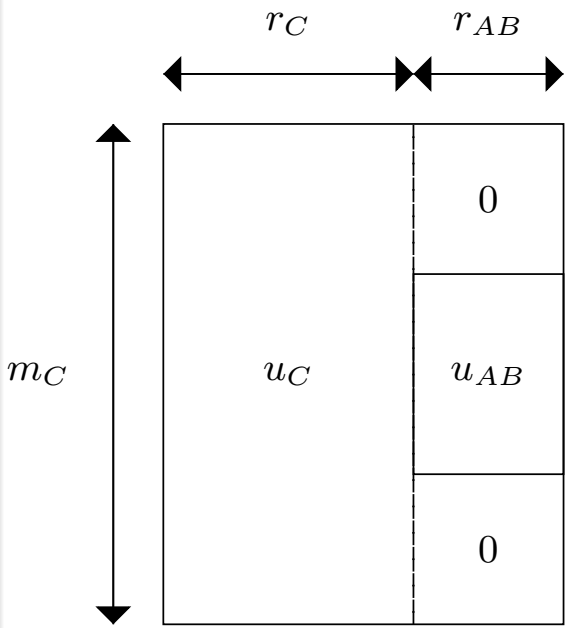
A low-rank structure $u_C v_C^t$ receives a low-rank contribution $u_{AB} v_{AB}^t$.

Algorithm

$$A = u_C v_C^t + u_{AB} v_{AB}^t = \left([u_C, u_{AB}]\right) \times \left([v_C, v_{AB}]\right)^t$$

- QR: $[u_C, u_{AB}] = Q_1 R_1$ $\Theta(m(r_C + r_{AB})^2)$
- QR: $[v_C, v_{AB}] = Q_2 R_2$ $\Theta(n(r_C + r_{AB})^2)$
- SVD: $R_1 R_2^t = u \sigma v^t$ $\Theta((r_C + r_{AB})^3)$

$$A = \left(Q_1 u \sigma\right) \times \left(Q_2 v\right)^t$$



Optimizations using RRQR to reduce complexity to $\Theta(n(r_C + r_{AB})r_C^*)$

Comparison of both strategies

Memory consumption

- **Minimal Memory** strategy really saves memory
- **Just-In-Time** strategy reduces the size of L' factors, but supernodes are allocated dense at the beginning: no gain in pure **right-looking**

Low-Rank extend-add

- **Minimal Memory** strategy requires expensive extend-add algorithms to update (recompress) low-rank structures with the **LR2LR** kernel
- **Just-In-Time** strategy continues to apply dense update at a smaller cost through the **LR2GE** kernel

3

Numerical Experiments

Experimental setup

Machine: 2 INTEL Xeon $E5-2680v3$ at 2.50 GHz

- 128 GB
- 24 cores

3D Matrices from The SuiteSparse Matrix Collection

- **Audi**: structural problem ($943\,695$ DoFs)
- **Atmosmodj**: atmospheric model ($1\,270\,432$ DoFs)
- **Geo1438**: geomechanical model of earth ($1\,437\,960$ DoFs)
- **Hook**: model of a steel hook ($1\,498\,023$ DoFs)
- **Serena**: gas reservoir simulation ($1\,391\,349$ DoFs)
- + Laplacian: Poisson problem (7-points stencil)

Parallelism is obtained with PASTIX static scheduling for multi-threaded architectures

Parameters

Entry parameters

- Tolerance τ: absolute parameter (normalized for each block)
- Compression method: SVD or RRQR
- Compression strategy: **Minimal Memory** or **Just-In-Time**
- Blocking sizes: between 128 and 256 in following experiments
- Compression sizes: 128 for width and 20 for height

Strategy **Minimal Memory**

- Blocks are compressed at the beginning
- Each contribution implies a recompression

Strategy **Just-In-Time**

- Blocks are compressed just before a supernode is eliminated
- Those blocks are never uncompressed

Costs distribution on the Atmosmodj matrix with $\tau = 10^{-8}$

	Complete	**Just-In-Time**	**Minimal Memory**
		RRQR	RRQR
Factorization time (s)			
Compression	-	3.4e+01	5.6+00
Block factorization (GETRF)	7.2e-01	7.3e-01	7.6e-01
Panel solve (TRSM)	1.7e+01	7.4e+00	7.9e+00
Update			
Formation of contribution	-	-	4.2e+01
Addition of contribution	-	-	7.3e+02
Dense udpate (GEMM)	4.6e+02	9.7e+01	2.4e+01
Total	**4.7e+02**	**1.4e+02**	**8.1e+02**
Solve time (s)	6.3e+00	3.0e+00	3.2e+00
Factors final size (GB)	16.3	7.49	7.31
Memory peak (GB)	16.3	16.3	7.31

Costs distribution on the Atmosmodj matrix with $\tau = 10^{-8}$

	Complete	**Just-In-Time**	**Minimal Memory**
		RRQR	RRQR
Factorization time (s)			
Compression	-	3.4e+01	5.6+00
Block factorization (GETRF)	7.2e-01	7.3e-01	7.6e-01
Panel solve (TRSM)	1.7e+01	7.4e+00	7.9e+00
Update			
Formation of contribution	-	-	4.2e+01
Addition of contribution	-	-	7.3e+02
Dense udpate (GEMM)	4.6e+02	9.7e+01	2.4e+01
Total	**4.7e+02**	**1.4e+02**	**8.1e+02**
Solve time (s)	6.3e+00	3.0e+00	3.2e+00
Factors final size (GB)	16.3	7.49	7.31
Memory peak (GB)	16.3	16.3	7.31

Costs distribution on the Atmosmodj matrix with $\tau = 10^{-8}$

	Complete	**Just-In-Time**	**Minimal Memory**
		RRQR	RRQR
Factorization time (s)			
Compression	-	3.4e+01	5.6+00
Block factorization (GETRF)	7.2e-01	7.3e-01	7.6e-01
Panel solve (TRSM)	1.7e+01	7.4e+00	7.9e+00
Update			
Formation of contribution	-	-	4.2e+01
Addition of contribution	-	-	7.3e+02
Dense udpate (GEMM)	4.6e+02	9.7e+01	2.4e+01
Total	**4.7e+02**	**1.4e+02**	**8.1e+02**
Solve time (s)	6.3e+00	3.0e+00	3.2e+00
Factors final size (GB)	16.3	7.49	7.31
Memory peak (GB)	16.3	16.3	7.31

Costs distribution on the Atmosmodj matrix with $\tau = 10^{-8}$

	Complete	**Just-In-Time**	**Minimal Memory**
		RRQR	RRQR
Factorization time (s)			
Compression	-	3.4e+01	5.6+00
Block factorization (GETRF)	7.2e-01	7.3e-01	7.6e-01
Panel solve (TRSM)	1.7e+01	7.4e+00	7.9e+00
Update			
Formation of contribution	-	-	4.2e+01
Addition of contribution	-	-	7.3e+02
Dense udpate (GEMM)	4.6e+02	9.7e+01	2.4e+01
Total	**4.7e+02**	**1.4e+02**	**8.1e+02**
Solve time (s)	6.3e+00	3.0e+00	3.2e+00
Factors final size (GB)	16.3	7.49	7.31
Memory peak (GB)	16.3	16.3	7.31

Costs distribution on the Atmosmodj matrix with $\tau = 10^{-8}$

	Complete	Just-In-Time	Minimal Memory
		RRQR	RRQR
Factorization time (s)			
Compression	-	3.4e+01	5.6+00
Block factorization (GETRF)	7.2e-01	7.3e-01	7.6e-01
Panel solve (TRSM)	1.7e+01	7.4e+00	7.9e+00
Update			
Formation of contribution	-	-	4.2e+01
Addition of contribution	-	-	7.3e+02
Dense udpate (GEMM)	4.6e+02	9.7e+01	2.4e+01
Total	**4.7e+02**	**1.4e+02**	**8.1e+02**
Solve time (s)	6.3e+00	3.0e+00	3.2e+00
Factors final size (GB)	16.3	7.49	7.31
Memory peak (GB)	16.3	16.3	7.31

Costs distribution on the Atmosmodj matrix with $\tau = 10^{-8}$

	Complete	Just-In-Time	Minimal Memory
		RRQR	RRQR
Factorization time (s)			
Compression	-	3.4e+01	5.6+00
Block factorization (GETRF)	7.2e-01	7.3e-01	7.6e-01
Panel solve (TRSM)	1.7e+01	7.4e+00	7.9e+00
Update			
Formation of contribution	-	-	4.2e+01
Addition of contribution	-	-	7.3e+02
Dense udpate (GEMM)	4.6e+02	9.7e+01	2.4e+01
Total	**4.7e+02**	**1.4e+02**	**8.1e+02**
Solve time (s)	6.3e+00	3.0e+00	3.2e+00
Factors final size (GB)	16.3	7.49	7.31
Memory peak (GB)	16.3	16.3	7.31

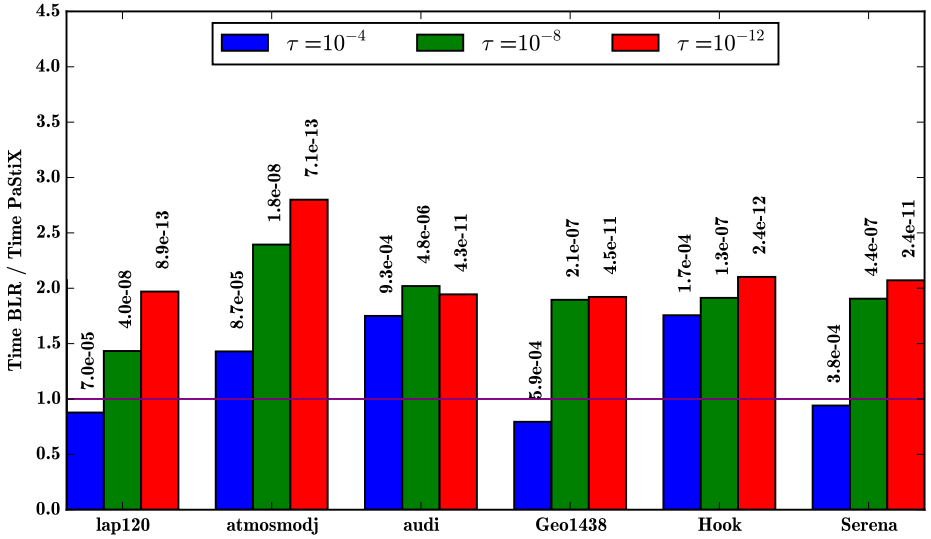
Costs distribution on the Atmosmodj matrix with $\tau = 10^{-8}$

	Complete	**Just-In-Time**	**Minimal Memory**
		RRQR	RRQR
Factorization time (s)			
Compression	-	3.4e+01	5.6+00
Block factorization (GETRF)	7.2e-01	7.3e-01	7.6e-01
Panel solve (TRSM)	1.7e+01	7.4e+00	7.9e+00
Update			
Formation of contribution	-	-	4.2e+01
Addition of contribution	-	-	7.3e+02
Dense udpate (GEMM)	4.6e+02	9.7e+01	2.4e+01
Total	**4.7e+02**	**1.4e+02**	**8.1e+02**
Solve time (s)	6.3e+00	3.0e+00	3.2e+00
Factors final size (GB)	16.3	7.49	7.31
Memory peak (GB)	16.3	16.3	7.31

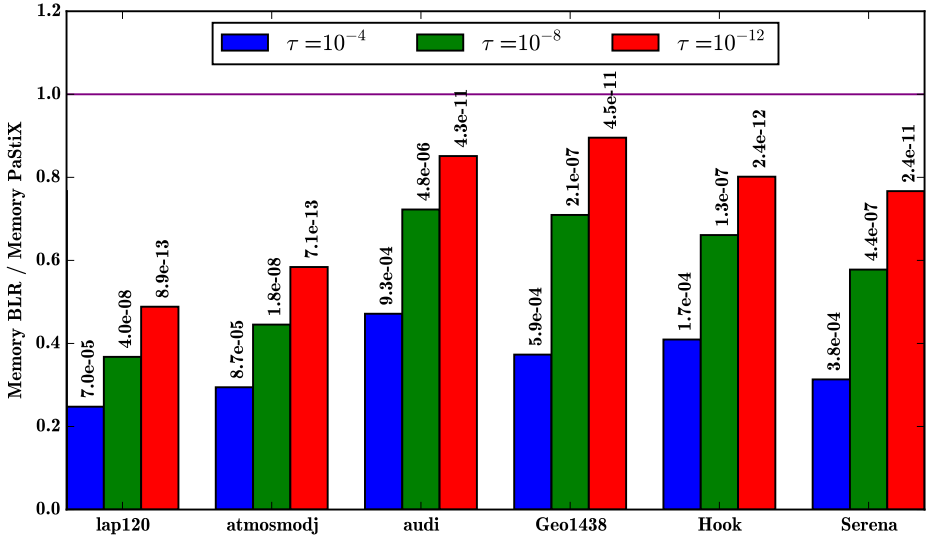
Costs distribution on the Atmosmodj matrix with $\tau = 10^{-8}$

	Complete	**Just-In-Time**	**Minimal Memory**
		RRQR	RRQR
Factorization time (s)			
Compression	-	3.4e+01	5.6+00
Block factorization (GETRF)	7.2e-01	7.3e-01	7.6e-01
Panel solve (TRSM)	1.7e+01	7.4e+00	7.9e+00
Update			
Formation of contribution	-	-	4.2e+01
Addition of contribution	-	-	7.3e+02
Dense udpate (GEMM)	4.6e+02	9.7e+01	2.4e+01
Total	**4.7e+02**	**1.4e+02**	**8.1e+02**
Solve time (s)	6.3e+00	3.0e+00	3.2e+00
Factors final size (GB)	16.3	7.49	7.31
Memory peak (GB)	16.3	16.3	7.31

Behaviour of RRQR/Minimal Memory

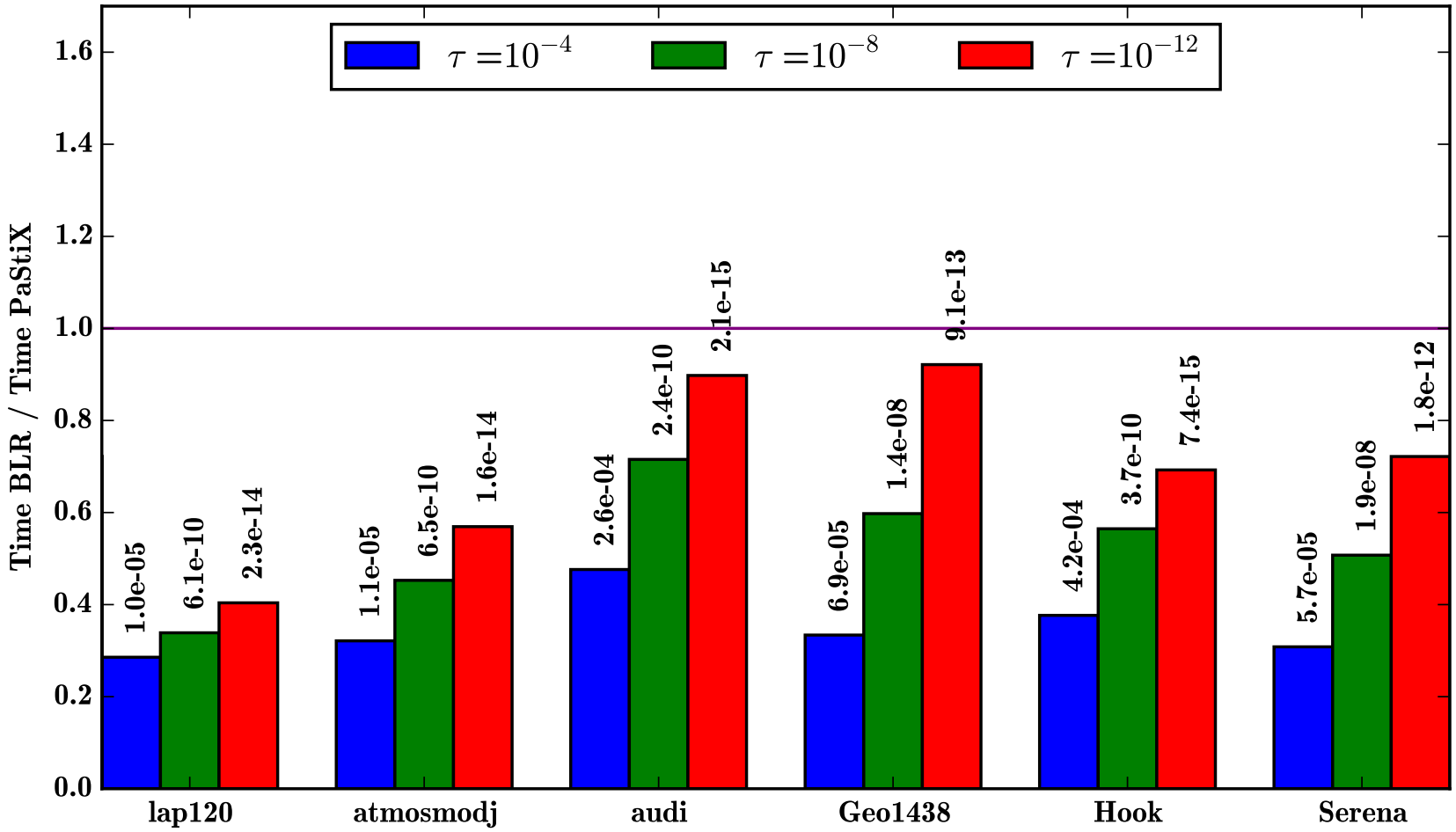


Performance

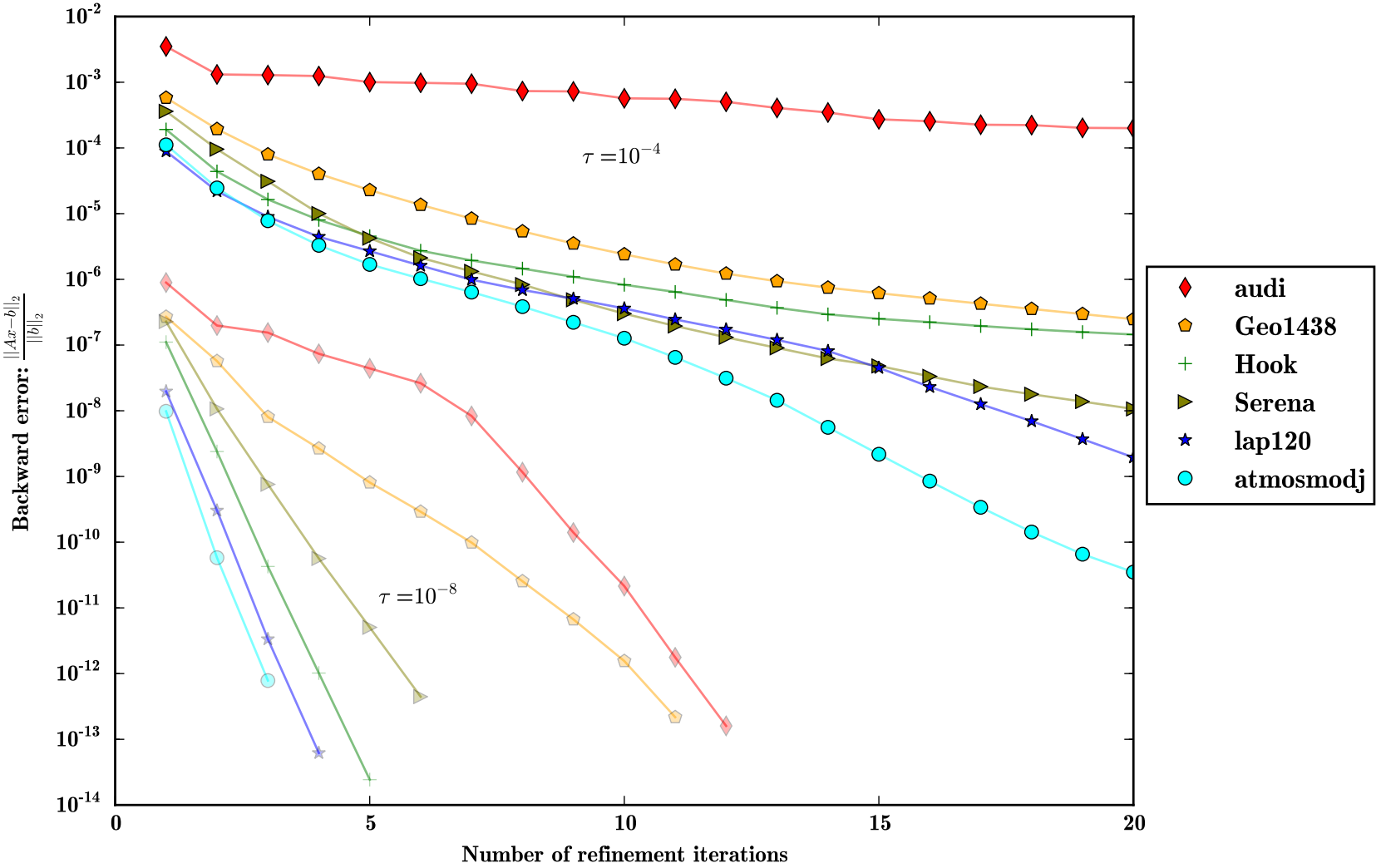


Memory footprint

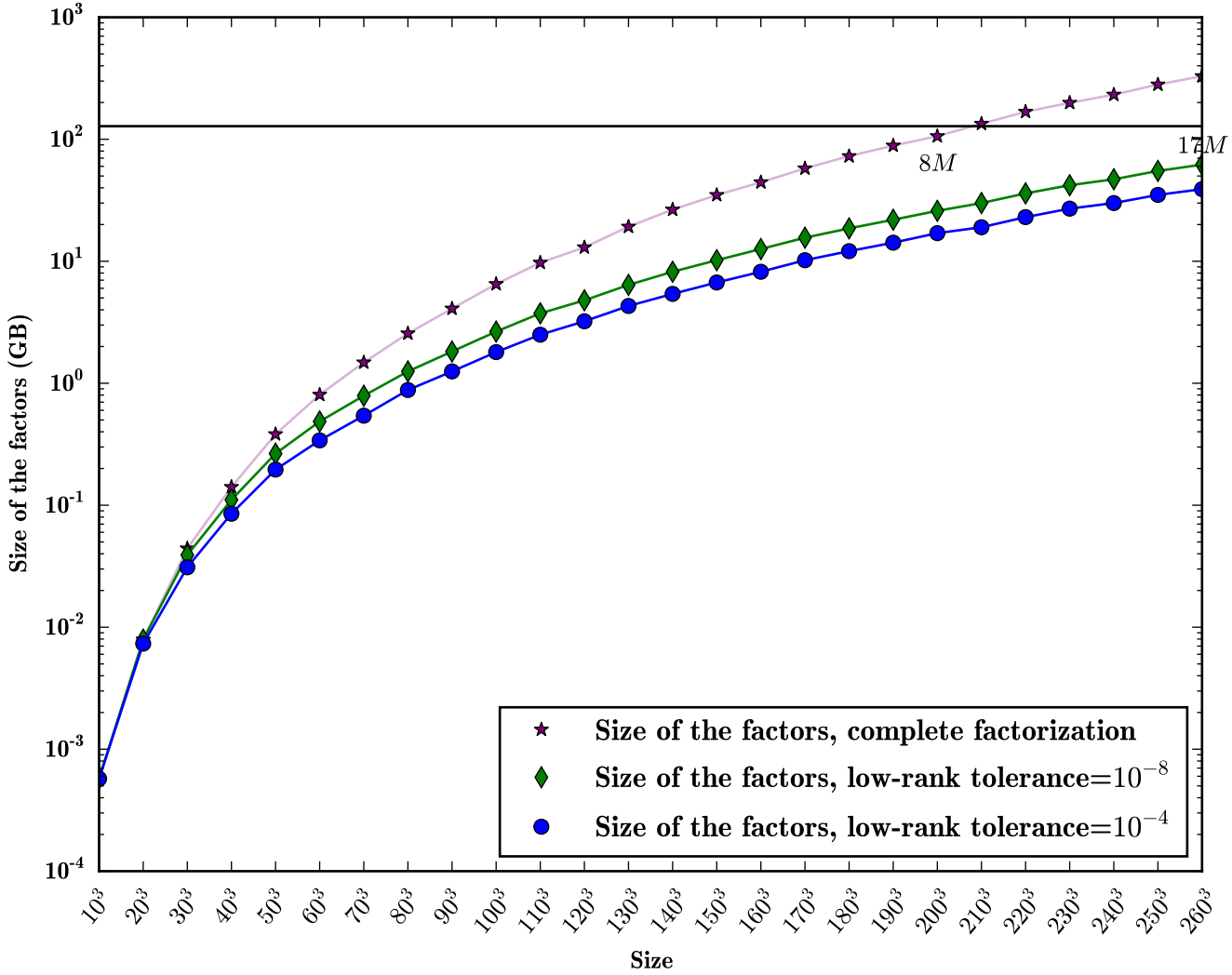
Performance of RRQR/Just-In-Time



Convergence of RRQR/Minimal Memory



Scaling on Laplacians: size of the factors using RRQR/Minimal Memory



4

Case study: PaStiX inside a HDG method

HORSE

Method

- Introduce an hybrid variable to solve a system smaller than the one obtained through DG
- Perform Domain-Decomposition to solve this system
- Use a direct solver for subdomains

Solve 3D frequency-domain Maxwell's equations

- $\mathbf{i}\omega\varepsilon_r\mathbf{E} - \mathbf{curl}\,\mathbf{H} = -\mathbf{J},$ in Ω
- $\mathbf{i}\omega\mu_r\mathbf{H} + \mathbf{curl}\,\mathbf{E} = 0,$ in Ω
- $\mathbf{n} \times \mathbf{E} = 0,$ on Γ_m
- $\mathbf{n} \times \mathbf{E} + \mathbf{n} \times (\mathbf{n} \times \mathbf{H}) = \mathbf{n} \times \mathbf{E}^{\text{inc}} + \mathbf{n} \times (\mathbf{n} \times \mathbf{H}^{\text{inc}}),$ on Γ_a

The challenge with DG for frequency-domain problems

Classical DG formulation

- Naturally adapted to heterogeneous media and discontinuous solutions
- Can easily deal with unstructured, possibly non-conforming meshes (h-adaptivity)
- High order with compact stencils and non-conforming approximations (p-adaptivity)
- Usually rely on polynomial interpolation but can also accommodate alternative functions
  (e.g. plane waves)
- Well-suited for efficient parallelization
- But leads to larger problems compared to continuous finite element methods

Hybridizable DG method in 3D

Principles of a HDG formulation

- Keep the advantages of classical DG methods
- Introduce an hybrid variable to decouple local problems defined at the element level
- Solve a reduced linear system for the hybrid variable unknowns only

Complexity: number of globally coupled degrees of freedom

- Classical DG method with $\mathbb{P}_p$ interpolation

$$(p+1)(p+2)(p+3)N_e \quad , \quad N_e \text{ is the \# of elements}$$

- HDG method with $\mathbb{P}_p$ interpolation

$$(p+1)(p+2)N_f \quad , \quad N_f \text{ is the \# of faces}$$

- Continuous finite element formulation based on Nedelec's first family of face/edge elements in a simplex (tetrahedron): $\frac{p(p+2)(p+3)}{2} N_e$

Hybridizable DG method in 3D

ANR TECSER project (May 2014 - April 2017)
Funded by DGA (French armament procurement agency)
http://www-sop.inria.fr/nachos/projects/tecser

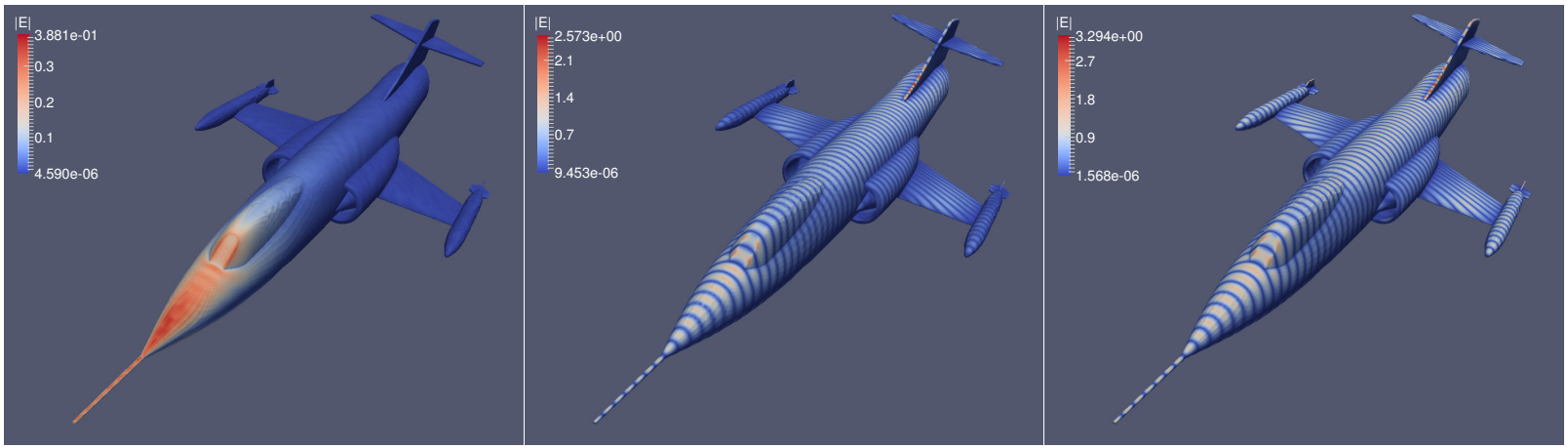
- Implementation of HDG for arbitrary high order interpolation
- Local definition (element-wise and face-wise) of the interpolation degree
- Extension of the formulation to a non-conforming hybrid hexahedral/tetrahedral mesh
- Scalability improvement
  PDE-based Schwartz domain decomposition algorithm
  PaStiX as a local (subdomain) solver
- Coupling with a BEM for an accurate treatment of far field radiation
- HORSE software
  High Order solver for Radar cross Section Evaluation

Hybridizable DG method in 3D

Scattering of a plane wave by a jet

- Unstructured tetrahedral mesh: $1,645,874$ elements / $3,521,251$ faces
- Frequency: 600 MHz, Wavelength: $\lambda \simeq 0.5$ m, Penalty parameter: $\tau = 1$

HDG method	# DoF $\mathbf{\Lambda}$ field	# DoF EM field
HDG-$\mathbb{P}_1$	21,127,506	39,500,976
HDG-$\mathbb{P}_2$	42,255,012	98,752,440
HDG-$\mathbb{P}_3$	70,425,020	197,504,880



Contour line of $|\mathbf{E}|$ - HDG$-\mathbb{P}_1$ to HDG$-\mathbb{P}_3$

Practical example

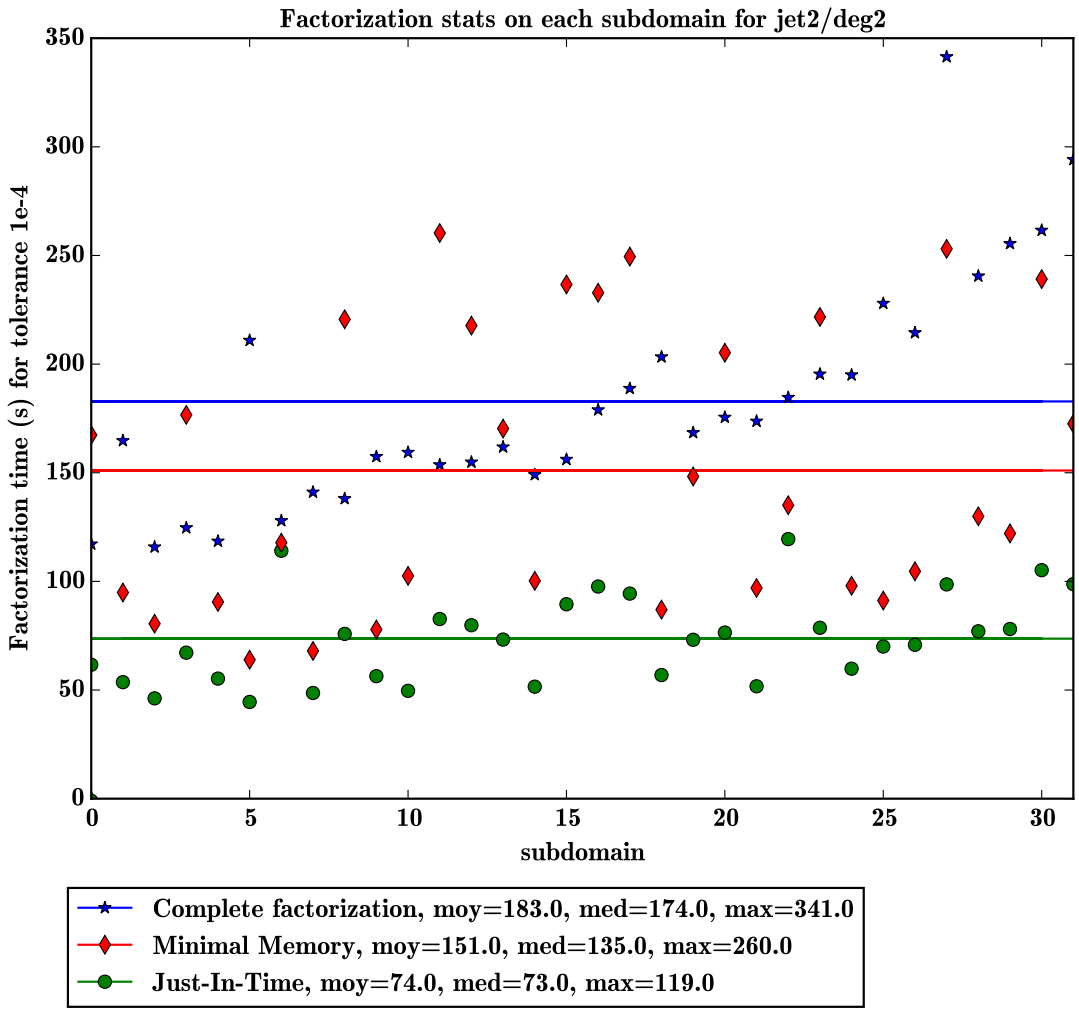
Study case

- Jet with $\mathbb{P}_2$ interpolation
- $42\ 255\ 012$ unknowns for $\mathbf{\Lambda}$
- A single factorization is performed on each subdomain
- There may be several solves for iterative refinement between subdomains

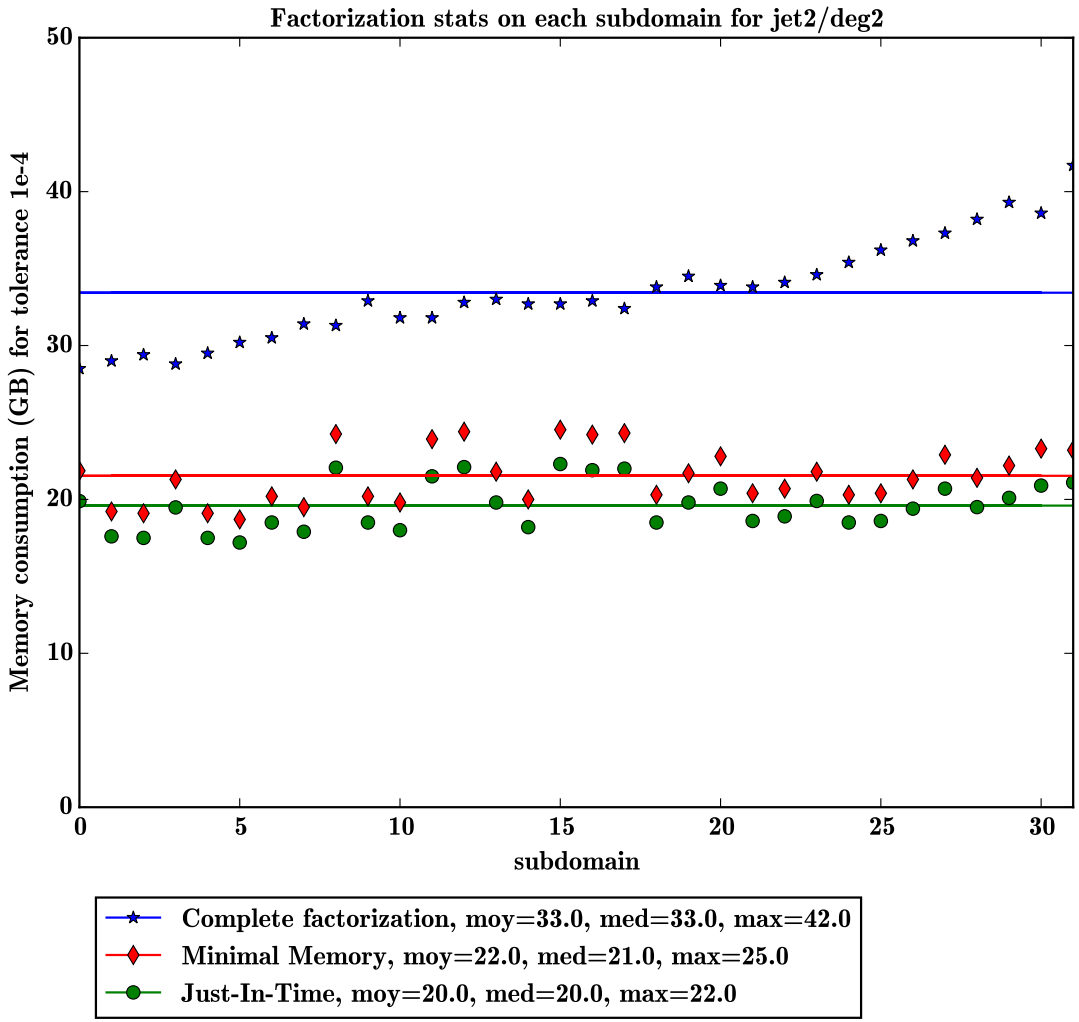
Machine

- Occigen: 24-cores nodes with 128 GB
- Each MPI process is hold on a socket with 12 cores
- In next experiments, we use 32, 48 or 64 subdomains
- Subdomains are ordered accordingly to the number of operations for the complete factorization

Factorization on each subdomain – 32 processes with $\tau = 1e-4$



Memory Consumption on each subdomain – 32 processes with $\tau = 1e-4$



Impact on HORSE

Subs	τ	Method	Fact(s)	Nb. of iters	Refinement (s)	HDGM (s)	Memory (GB)
32	-	Complete-Factorization	526.0	9	254.5	814.0	41.7
	1e-4	Just-In-Time	209.3	9	164.8	405.8	41.7 (22.3)
		Minimal-Memory	516.7	15	273.2	822.0	24.5
	1e-8	Just-In-Time	325.2	9	192.9	550.4	41.7 (29.4)
		Minimal-Memory	600.1	9	193.2	825.8	30.5
48	-	Complete-Factorization	237.3	8	118.6	374.6	24.9
	1e-4	Just-In-Time	112.2	9	106.1	237.1	24.9 (14.1)
		Minimal-Memory	229.3	13	159.7	407.5	15.3
	1e-8	Just-In-Time	171.5	8	105.8	296.1	24.9 (18.1)
		Minimal-Memory	319.4	8	109.8	447.9	18.8
64	-	Complete-Factorization	179.8	9	104.7	298.3	17.4
	1e-4	Just-In-Time	79.7	10	91.1	184.1	17.4 (10.0)
		Minimal-Memory	138.1	13	120.7	272.2	11.0
	1e-8	Just-In-Time	124.6	9	90.0	228.2	17.4 (12.9)
		Minimal-Memory	239.2	9	91.5	344.5	13.7

Conclusion

Block Low-Rank solver

- Exploits the symbolic structure
- Demonstrate the extend-add issues in a supernodal context
- The version presented follows the parallelism of PaStiX
- On-going work to implement efficiently this approach over runtime systems

PaStiX in HORSE

- Low-rank compression is useful to reduce memory footprint and/or time-to-solution
- Future work: study load imbalance between subdomains
- Future work: impact with $\mathbb{P}_3/\mathbb{P}_4$ interpolation

PaStiX 6.0.0alpha is now available!

`http://gitlab.inria.fr/solverstack/pastix`

- Support shared memory with different schedulers:
  - sequential
  - static scheduler
  - PaRSEC runtime system
- Low-rank support
- Cholesky and LU factorizations

Thank you.